Today's Party

- DDLs
- Complex Joins
- Views
- Nested Subqueries
- Triggers
- Database Application Example

Example Database

<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
<th>login</th>
<th>age</th>
<th>gpa</th>
</tr>
</thead>
<tbody>
<tr>
<td>53666</td>
<td>Faloutsos</td>
<td>christos@cs</td>
<td>45</td>
<td>4.0</td>
</tr>
<tr>
<td>53688</td>
<td>Bieber</td>
<td>jbieber@cs</td>
<td>21</td>
<td>3.9</td>
</tr>
<tr>
<td>53677</td>
<td>Tupac</td>
<td>shakur@cs</td>
<td>26</td>
<td>3.5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sid</th>
<th>cid</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>53831</td>
<td>Pilates101</td>
<td>C</td>
</tr>
<tr>
<td>53688</td>
<td>Reggae203</td>
<td>D</td>
</tr>
<tr>
<td>53688</td>
<td>Topology112</td>
<td>A</td>
</tr>
<tr>
<td>53666</td>
<td>Massage105</td>
<td>D</td>
</tr>
</tbody>
</table>
Table Definition (DDL)

```
CREATE TABLE <table-name>

[column-definition]*
[constraints]*
} [table-options];
```

- **Column-Definition**: Comma separated list of column names with types.
- **Constraints**: Primary key, foreign key, and other meta-data attributes of columns.
- **Table-Options**: DBMS-specific options for the table (not SQL-92).

Table Definition Example

```
CREATE TABLE student (
    sid INT,
    name VARCHAR(16),
    login VARCHAR(32),
    age SMALLINT,
    gpa FLOAT
);

CREATE TABLE enrolled (
    sid INT,
    cid VARCHAR(32),
    grade CHAR(1)
);
```

Common Data Types

- **CHAR(n)**, **VARCHAR(n)**
- **TINYINT**, **SMALLINT**, **INTEGER**, **BIGINT**
- **NUMERIC(p,d)**, **FLOAT**, **DOUBLE**, **REAL**
- **DATE**, **TIME**
- **BINARY(n)**, **VARBINARY(n)**, **BLOB**
Comment About BLOBs

- Don’t store large files in your database!
- Put the file on the filesystem and store a URI in the database.
- Many app frameworks will do this automatically for you.
- More information:
  - *To BLOB or Not to BLOB: Large Object Storage in a Database or a Filesystem?*

Useful Non-standard Types

- TEXT
- BOOLEAN
- ARRAY
- Some systems also support user-defined types.

Integrity Constraints

```sql
CREATE TABLE student ( sid INT PRIMARY KEY, name VARCHAR(16), login VARCHAR(32) UNIQUE, age SMALLINT CHECK (age > 0), gpa FLOAT );
CREATE TABLE enrolled ( sid INT REFERENCES student (sid), cid VARCHAR(32) NOT NULL, grade CHAR(1), PRIMARY KEY (sid, cid) );
```
Primary Keys

- Single-column primary key:

```sql
CREATE TABLE student (          
  sid  INT PRIMARY KEY,        
  ...                           
)
```

- Multi-column primary key:

```sql
CREATE TABLE student (          
  ...                           
  PRIMARY KEY (sid, name)      
)
```

Foreign Key References

- Single-column reference:

```sql
CREATE TABLE enrolled (            
  sid  INT REFERENCES student (sid),  
  ...                                 
)
```

- Multi-column reference:

```sql
CREATE TABLE enrolled (            
  ...                                 
  FOREIGN KEY (sid, ...) REFERENCES student (sid, ...) 
)
```

You can define what happens when the parent table is modified:
- `CASCADE`
- `RESTRICT`
- `NO ACTION`
- `SET NULL`
- `SET DEFAULT`
Foreign Key References

- Delete/update the enrollment information when a student is changed:

```sql
CREATE TABLE enrolled (  
    FOREIGN KEY (sid)  
    REFERENCES student (sid)  
    ON DELETE CASCADE  
    ON UPDATE CASCADE
```

Value Constraints

- Ensure one-and-only-one value exists:

```sql
CREATE TABLE student (  
    login VARCHAR(32) UNIQUE,
```

- Make sure a value is not null:

```sql
CREATE TABLE enrolled (  
    cid VARCHAR(32) NOT NULL,
```

- Make sure that an expression evaluates to true for each row in the table:

```sql
CREATE TABLE enrolled (  
    age SMALLINT CHECK (age > 0),
```

- Can be expensive to evaluate, so tread lightly...
Auto-Generated Keys

- Automatically create a unique integer id for whenever a row is inserted \((last + 1)\).
- Implementations vary wildly:
  - SQL:2003 → `IDENTITY`
  - MySQL → `AUTO_INCREMENT`
  - Postgres → `SERIAL`
  - SQL Server → `SEQUENCE`
  - DB2 → `SEQUENCE`
  - Oracle → `SEQUENCE`

```sql
CREATE TABLE student (
  sid INT PRIMARY KEY AUTO_INCREMENT,
  ...)
```

```sql
INSERT INTO student
  (sid, name, login, age, gpa)
VALUES
  (NULL, "Christos", "@cs", 45, 4.0);
```

Conditional Table Creation

- `IF NOT EXISTS` prevents the DBMS from trying to create a table twice.

```sql
CREATE TABLE IF NOT EXISTS student (
  sid INT PRIMARY KEY,
  name VARCHAR(16),
  login VARCHAR(32) UNIQUE,
  age SMALLINT CHECK (age > 0),
  gpa FLOAT
);```
Dropping Tables

- Completely removes a table from the database. Deletes everything related to the table (e.g., indexes, views, triggers, etc):

```sql
DROP TABLE student;
```

- Can also use **IF EXISTS** to avoid errors:

```sql
DROP TABLE IF EXISTS student;
```

Modifying Tables

- SQL lets you add/drop columns in a table after it is created:

```sql
ALTER TABLE student
ADD COLUMN phone VARCHAR(32) NOT NULL;
```

```sql
ALTER TABLE student
DROP COLUMN login;
```

- This is really expensive!!! Tread lightly....

Modifying Tables

- You can also modify existing columns (rename, change type, change defaults, etc):

```sql
ALTER TABLE student
ALTER COLUMN name TYPE VARCHAR(32);
```

```sql
ALTER TABLE student
CHANGE COLUMN name name VARCHAR(32);
```
Accessing Table Schema

- You can query the DBMS’s internal `INFORMATION_SCHEMA` catalog to get info about the database.
- ANSI standard set of read-only views that provide info about all of the tables, views, columns, and procedures in a database.
- Every DBMS also have non-standard shortcuts to do this.

List all of the tables in the current database:

- **Postgres**
  ```sql
  \d;  
  ```
- **MySQL**
  ```sql
  SHOW TABLES;  
  ```
- **SQLite**
  ```sql
  .tables;  
  ```

List the column info for the student table:

- **Postgres**
  ```sql
  DESCRIBE student;  
  ```
- **MySQL**
  ```sql
  .schema student;  
  ```
- **SQLite**
  ```sql
  ```
Today's Party

- DDLs
- Complex Joins
- Views
- Nested Subqueries
- Triggers
- Database Application Example

Example Database

<table>
<thead>
<tr>
<th>STUDENT</th>
<th>ENROLLED</th>
</tr>
</thead>
<tbody>
<tr>
<td>sid</td>
<td>name</td>
</tr>
<tr>
<td>53666</td>
<td>Pilates</td>
</tr>
<tr>
<td>53688</td>
<td>Bieber</td>
</tr>
<tr>
<td>53655</td>
<td>Tupac</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>COURSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>cid</td>
</tr>
<tr>
<td>-----</td>
</tr>
<tr>
<td>Pilates101</td>
</tr>
<tr>
<td>Reggae203</td>
</tr>
<tr>
<td>Topology112</td>
</tr>
<tr>
<td>Massage105</td>
</tr>
</tbody>
</table>

Join Query Grammar

```sql
SELECT ... FROM table-name1 join-type table-name2 ON qualification [WHERE ...]
```

- **Join-Type**: The type of join to compute.
- **Qualification**: Expression that determines whether a tuple from table1 can be joined with table2. Comparison of attributes or constants using operators =, ≠, <, >, ≤, and ≥.
### INNER JOIN

**SELECT** `name, cid, grade`  
**FROM** `student`  
**INNER JOIN** `enrolled`  
**ON** `student.sid = enrolled.sid`

<table>
<thead>
<tr>
<th>name</th>
<th>cid</th>
<th>Grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bieber</td>
<td>Reggae203</td>
<td>D</td>
</tr>
<tr>
<td>Faloutsos</td>
<td>Pilates101</td>
<td>C</td>
</tr>
<tr>
<td>Faloutsos</td>
<td>Topology112</td>
<td>A</td>
</tr>
<tr>
<td>Tupac</td>
<td>Massage105</td>
<td>D</td>
</tr>
<tr>
<td>Faloutsos</td>
<td>Pilates101</td>
<td>C</td>
</tr>
</tbody>
</table>

### OUTER JOIN

**SELECT** `student.sid, cid, grade`  
**FROM** `student`  
**LEFT OUTER JOIN** `enrolled`  
**ON** `student.sid = enrolled.sid`

<table>
<thead>
<tr>
<th>name</th>
<th>cid</th>
<th>Grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bieber</td>
<td>Reggae203</td>
<td>D</td>
</tr>
<tr>
<td>Faloutsos</td>
<td>Pilates101</td>
<td>C</td>
</tr>
<tr>
<td>Faloutsos</td>
<td>Topology112</td>
<td>A</td>
</tr>
<tr>
<td>Tupac</td>
<td>Massage105</td>
<td>D</td>
</tr>
<tr>
<td>Faloutsos</td>
<td>Pilates101</td>
<td>C</td>
</tr>
<tr>
<td>Tupac</td>
<td>NULL</td>
<td>NULL</td>
</tr>
</tbody>
</table>
**OUTER JOIN**

```
SELECT student.sid, cid, grade
FROM enrolled RIGHT OUTER JOIN student
ON student.sid = enrolled.sid
```

<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
<th>login</th>
<th>age</th>
<th>gpa</th>
</tr>
</thead>
<tbody>
<tr>
<td>53666</td>
<td>Faloutsos</td>
<td>christos@cs</td>
<td>45</td>
<td>4.0</td>
</tr>
<tr>
<td>53688</td>
<td>Bieber</td>
<td>jbieber@cs</td>
<td>21</td>
<td>3.9</td>
</tr>
<tr>
<td>53677</td>
<td>Tupac</td>
<td>shakur@cs</td>
<td>36</td>
<td>3.5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sid</th>
<th>cid</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>53666</td>
<td>Pilates101</td>
<td>C</td>
</tr>
<tr>
<td>53688</td>
<td>Reggae203</td>
<td>D</td>
</tr>
<tr>
<td>53688</td>
<td>Topology112</td>
<td>A</td>
</tr>
<tr>
<td>53666</td>
<td>Massage105</td>
<td>D</td>
</tr>
</tbody>
</table>

**Join Types**

```
SELECT * FROM A JOIN B ON A.id = B.id
```

<table>
<thead>
<tr>
<th>Join Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>INNER JOIN</td>
<td>Join where A and B have same value</td>
</tr>
<tr>
<td>LEFT OUTER JOIN</td>
<td>Join where A and B have same value AND where only A has a value</td>
</tr>
<tr>
<td>RIGHT OUTER JOIN</td>
<td>Join where A and B have same value AND where only B has a value</td>
</tr>
<tr>
<td>FULL OUTER JOIN</td>
<td>Join where A and B have same value AND where A or B have unique values</td>
</tr>
<tr>
<td>CROSS JOIN</td>
<td>Cartesian Product</td>
</tr>
</tbody>
</table>

**Today's Party**

- DDLs
- Complex Joins
- Views
- Nested Subqueries
- Triggers
- Database Application Example
Views

• Creates a “virtual” table containing the output from a **SELECT** query.
• Mechanism for hiding data from view of certain users.
• Can be used to simplify a complex query that is executed often.
  – Won’t make it faster though!

View Example

• Create a view of the CS student records with just their id, name, and login.

```
CREATE VIEW CompSciStudentInfo AS
SELECT sid, name, login
FROM student
WHERE login LIKE '%@cs';
```

Original Table

<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
<th>login</th>
<th>age</th>
<th>gpa</th>
</tr>
</thead>
<tbody>
<tr>
<td>5366</td>
<td>Faloutsos</td>
<td>christos@cs</td>
<td>45</td>
<td>4.0</td>
</tr>
<tr>
<td>53688</td>
<td>Bieber</td>
<td>jbieber@cs</td>
<td>21</td>
<td>3.9</td>
</tr>
</tbody>
</table>

View

<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
<th>login</th>
</tr>
</thead>
<tbody>
<tr>
<td>5366</td>
<td>Faloutsos</td>
<td>christos@cs</td>
</tr>
<tr>
<td>53688</td>
<td>Bieber</td>
<td>jbieber@cs</td>
</tr>
</tbody>
</table>

View Example

• Create a view with the average age of the students enrolled in each course.

```
CREATE VIEW CourseAge AS
SELECT cid, AVG(age) AS avg_age
FROM student, enrolled
WHERE student.sid = enrolled.sid
GROUP BY enrolled.cid;
```

```
cid | avg_age |
-----|---------|
Massage105 | 45.0 |
Pilates101 | 45.0 |
Topology112 | 21.0 |
Reggae203 | 21.0 |
```
Views vs. SELECT INTO

```sql
CREATE VIEW AvgGPA AS
SELECT AVG(gpa) AS avg_gpa FROM student
WHERE login LIKE '%@cs'

SELECT AVG(gpa) AS avg_gpa INTO AvgGPA
FROM student WHERE login LIKE '%@cs'
```

- **INTO** → Creates static table that does not get updated when student gets updated.
- **VIEW** → Dynamic results are only materialized when needed.

Today's Party

- DDLs
- Complex Joins
- Views
- Nested Subqueries
- Triggers
- Database Application Example

 Nested Queries

- Queries containing other queries
  - "outer query"
  - "inner query"
- Inner query:
  - Can appear in `FROM` or `WHERE` clause

```sql
SELECT cname FROM borrower WHERE
  cname IN (SELECT cname FROM depositor)
```

Think of this as a function that returns the result of the inner query

<table>
<thead>
<tr>
<th>cname</th>
<th>Johnson</th>
<th>Smith</th>
<th>Jones</th>
<th>Smith</th>
</tr>
</thead>
</table>

Nested Queries

• Find the names of students in ‘Massage105’

```sql
SELECT name FROM student
WHERE...

"sid in the set of people that take Massage105"
```

Faloutsos/Pavlo
CMU SCS 15-415/615
42

Nested Queries

• Find the names of students in ‘Massage105’

```sql
SELECT name FROM student
WHERE...
SELECT sid FROM enrolled
WHERE cid = 'Massage105'
```

Faloutsos/Pavlo
CMU SCS 15-415/615
43

Nested Queries

• Find the names of students in ‘Massage105’

```sql
SELECT name FROM student
WHERE sid IN (
SELECT sid FROM enrolled
WHERE cid = 'Massage105'
)
```

Faloutsos/Pavlo
CMU SCS 15-415/615
44
Nested Queries

- **ALL** → Must satisfy expression for all rows in sub-query
- **ANY** → Must satisfy expression for at least one row in sub-query.
- **IN** → Equivalent to `=ANY()`.
- **EXISTS** → At least one row is returned.

  • Nested queries are difficult to optimize. Try to avoid them if possible.

Find the names of students in "Massage105":

```
SELECT name FROM student
WHERE sid = ANY(
    SELECT sid FROM enrolled
    WHERE cid = 'Massage105'
)
```

Find student record with the highest id.

- This won’t work in **SQL-92**:

  ```
  SELECT MAX(sid), name FROM student;
  ```

  - Runs in MySQL, but you get wrong answer:

<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>56688</td>
<td>Tupac</td>
</tr>
</tbody>
</table>
Nested Queries

• Find student record with the highest id.

```
SELECT sid, name FROM student
WHERE ...

"is greater than every other sid"
```


```
SELECT sid, name FROM student
WHERE sid is greater than every
SELECT sid FROM enrolled
```


```
SELECT sid, name FROM student
WHERE sid => ALL(
    SELECT sid FROM enrolled
)
```

<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>5568</td>
<td>Bieber</td>
</tr>
</tbody>
</table>
Nested Queries

• Find student record with the highest id.

```sql
SELECT sid, name FROM student
WHERE sid IN (
    SELECT MAX(sid) FROM enrolled
)
```

<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>53688</td>
<td>Bieber</td>
</tr>
</tbody>
</table>

Nested Queries

• Find all courses that nobody is enrolled in.

```sql
SELECT * FROM course
WHERE ...
```

"with no tuples in the ‘enrolled’ table"

<table>
<thead>
<tr>
<th>cid</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pilates101</td>
<td>Pilates</td>
</tr>
<tr>
<td>Reggae203</td>
<td>20th Century Reggae</td>
</tr>
<tr>
<td>Karate101</td>
<td>Karate Kid Aerobics</td>
</tr>
<tr>
<td>Topology112</td>
<td>Topology + Squirrels</td>
</tr>
<tr>
<td>Massage105</td>
<td>Massage &amp; Holistic Therapy</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sid</th>
<th>cid</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>53666</td>
<td>Pilates101</td>
<td>C</td>
</tr>
<tr>
<td>53688</td>
<td>Reggae203</td>
<td>D</td>
</tr>
<tr>
<td>53888</td>
<td>Karate101</td>
<td>A</td>
</tr>
<tr>
<td>53888</td>
<td>Topology112</td>
<td>A</td>
</tr>
<tr>
<td>53666</td>
<td>Massage105</td>
<td>D</td>
</tr>
</tbody>
</table>

Nested Queries

• Find all courses that nobody is enrolled in.

```sql
SELECT * FROM course
WHERE NOT EXISTS(
    SELECT * FROM enrolled
)
```
Nested Queries

- Find all courses that nobody is enrolled in.

```sql
SELECT * FROM course
WHERE NOT EXISTS(
    SELECT * FROM enrolled
    WHERE course.cid = enrolled.cid
)
```

Today's Party

- DDLs
- Complex Joins
- Views
- Nested Subqueries
- Triggers
- Database Application Example

Database Triggers

- Procedural code that is automatically executed in response to certain events on a particular table or view in a database.

**BEFORE/AFTER**
- INSERT
- UPDATE
- DELETE
Trigger Example

- Set a timestamp field whenever a row in the enrolled table is updated.
- First we need to add our timestamp field.

```
ALTER TABLE enrolled
ADD COLUMN updated TIMESTAMP;
```

Trigger Example

- Register a function that sets the ‘updated’ column with the current timestamp.

```
CREATE OR REPLACE FUNCTION update_col()
RETURNS TRIGGER AS $$
BEGIN
    NEW.updated = NOW();
    RETURN NEW;
END;
$$
language 'plpgsql';
```

Trigger Example

- Invoke the `update_col` function when a row in the enrolled table is updated.

```
CREATE TRIGGER update_enrolled_modtime
AFTER UPDATE ON enrolled
FOR EACH ROW
EXECUTE PROCEDURE update_col();
```

Postgres
Today's Party

• DDLs
• Complex Joins
• Views
• Nested Subqueries
• Triggers
  • Database Application Example

Outline of an DB application

• Establish connection with DB server
• Authenticate (user/password)
• Execute SQL statement(s)
• Process results
• Close connection
Sample Python Code

- [http://www.cs.cmu.edu/~christos/courses/dbms.S14/PYTHON-examples/csv2sql.py](http://www.cs.cmu.edu/~christos/courses/dbms.S14/PYTHON-examples/csv2sql.py)

- Or follow instructions at

```python
import sqlite

conn = sqlite.connect('test.db')

cursor = conn.cursor()
cursor.execute('create table if not exists test (name text, address text, state text, city text)')

for row in cursor:
    print(row)
```

```python
print("csv2sql inserted ", filename)
print("csv2sql inserted ", filename)
for row in cursor:
    print(row)
```
Cursors

- Used to iterate through the results of query.
- Enables result rows to be processed sequentially.
  - Runs counter to SQL’s set-based nature.
  - Very inefficient in most cases!
Outline of an SQL application:

- Establish connection with db server
- Authenticate (user/password)
- Execute SQL statement(s) (using **cursors**)
- Process results
- Close connection

**ORM Libraries**

- **Object-Relational Mapping**
- Automatically convert classes into database-backed objects.
- Method calls on objects are automatically converted into SQL queries.
- Removes the tediousness of writing SQL queries directly in application code.
ORM Example

```python
class Location(models.Model):
    zipcode = CharField(max_length=5, primary_key=True)
    state = USStateField()
    city = CharField(max_length=64)

class Company(models.Model):
    name = CharField(max_length=64, unique=True)
    address1 = CharField(max_length=128)
    location = ForeignKey(Location)
    website = URLField()
    public = BooleanField(default=True)
```

ORM Example

```
CREATE TABLE location (  
    zipcode VARCHAR(5) NOT NULL,
    state CHAR(2) NOT NULL,
    city VARCHAR(64) NOT NULL,
    PRIMARY KEY (zipcode),
);

CREATE TABLE company (  
    id INT(11) NOT NULL AUTO_INCREMENT,
    name VARCHAR(64) NOT NULL,
    address1 VARCHAR(128) NOT NULL,
    location_id VARCHAR(5) NOT NULL \  
    REFERENCES location (zipcode),
    website VARCHAR(200) NOT NULL,
    public TINYINT(1) NOT NULL,
    PRIMARY KEY (id),
);```

ORM Libraries

- **Standalone:**
  - Hibernate (Java)
  - SQLAlchemy (Python)
  - Doctrine (PHP)

- **Integrated:**
  - Django (Python)
  - ActiveRecord (Ruby on Rails)
  - CakePHP (PHP)